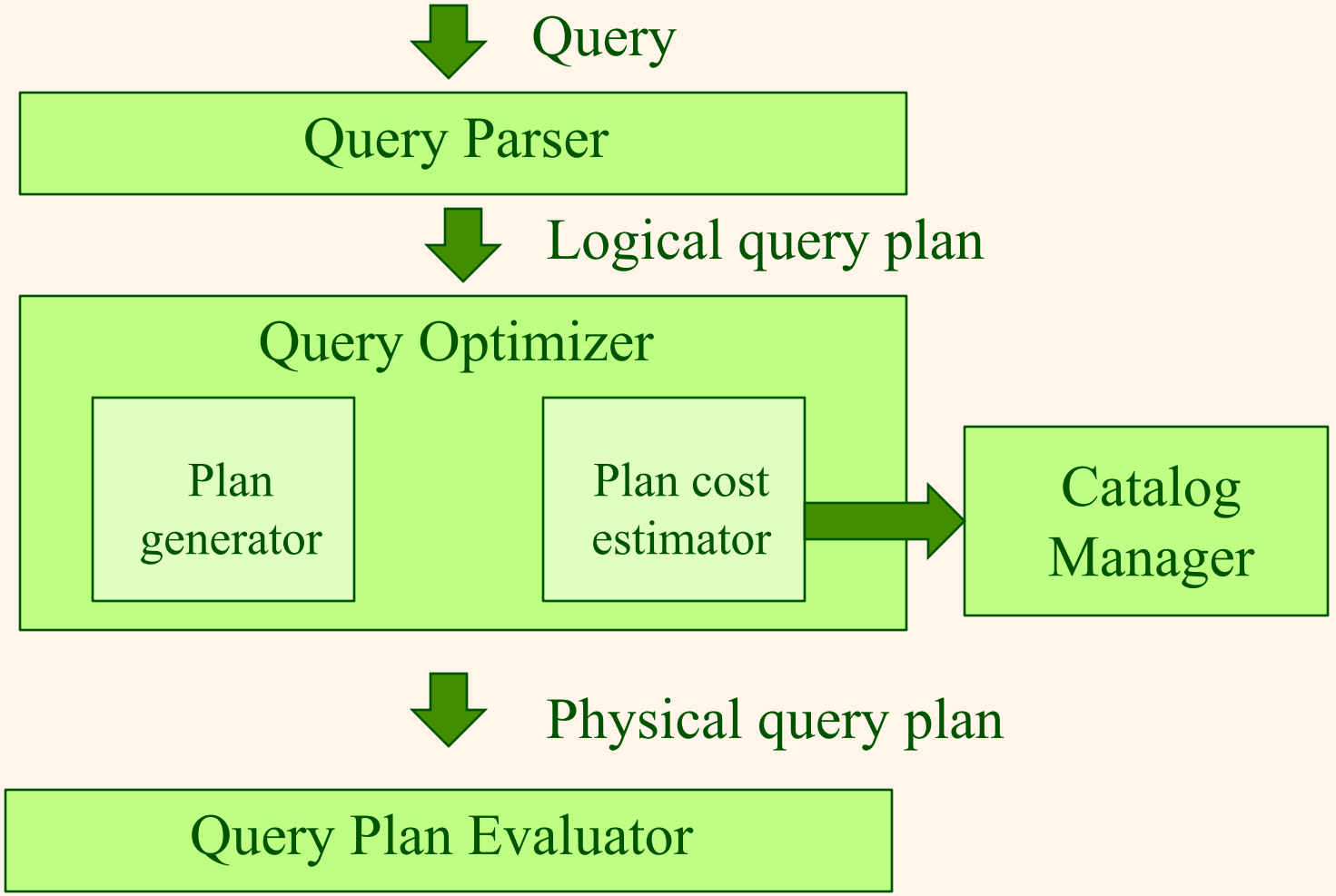
Query Optimization

Query optimization



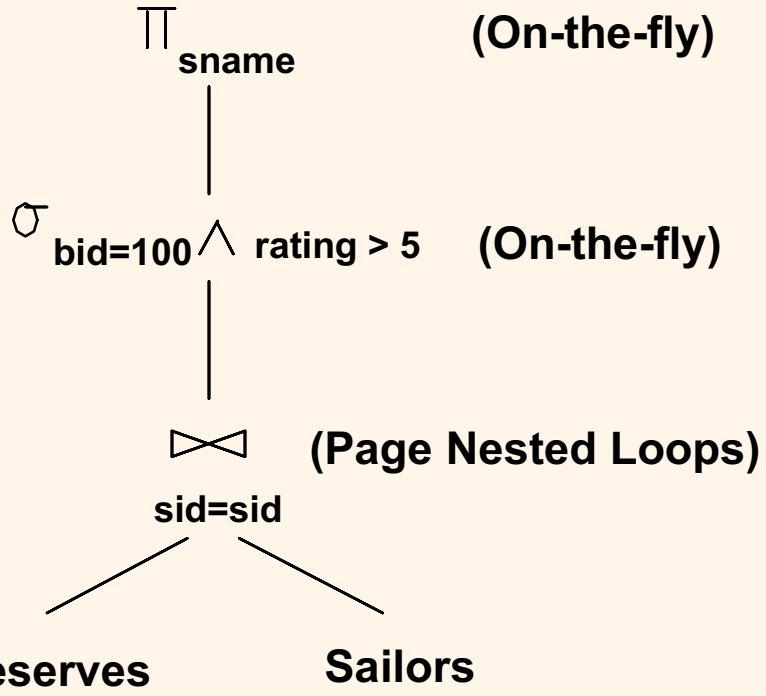
Two main questions

- How to generate (a good subset of) the possible plans?
- How to compute the cost of each plan?

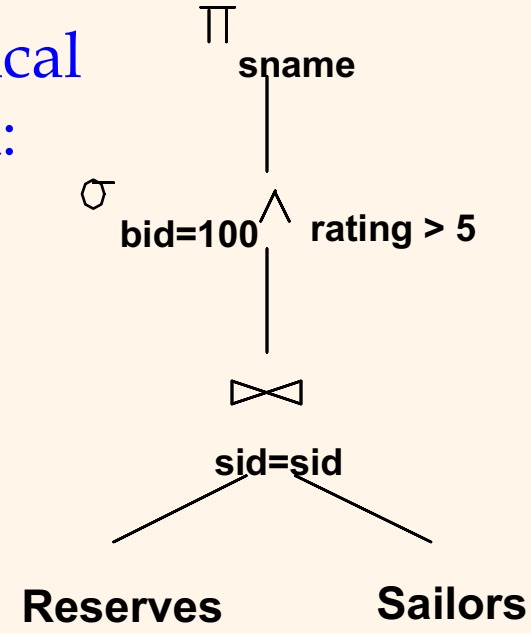
- Let's start with the second question….
  - Basically it's about putting together the per-operator calculations we have been doing already

A first physical query plan

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```



Logical plan:

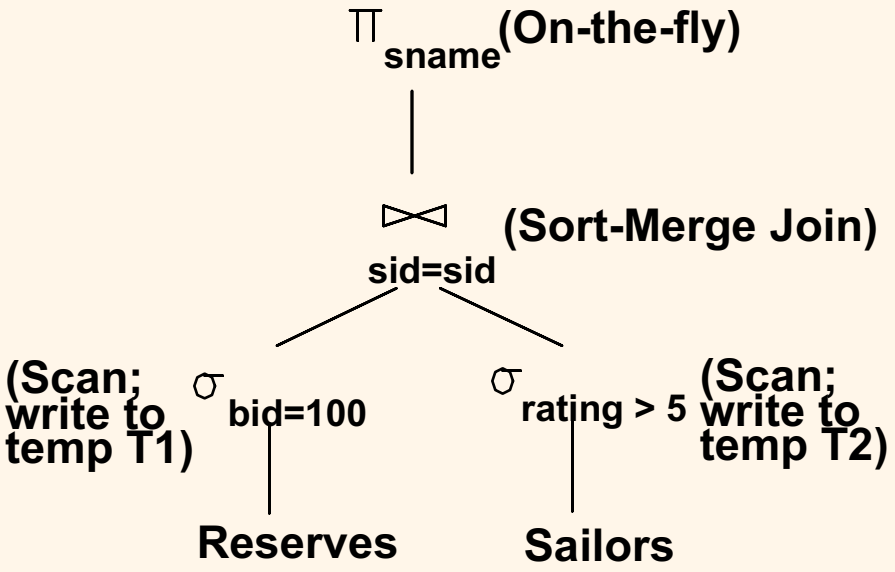


- Cost: 1000+500*1000 = 501,000 page I/Os
- Convention: **left child** of join = **outer relation**

FAQ: what does "on the fly" mean?

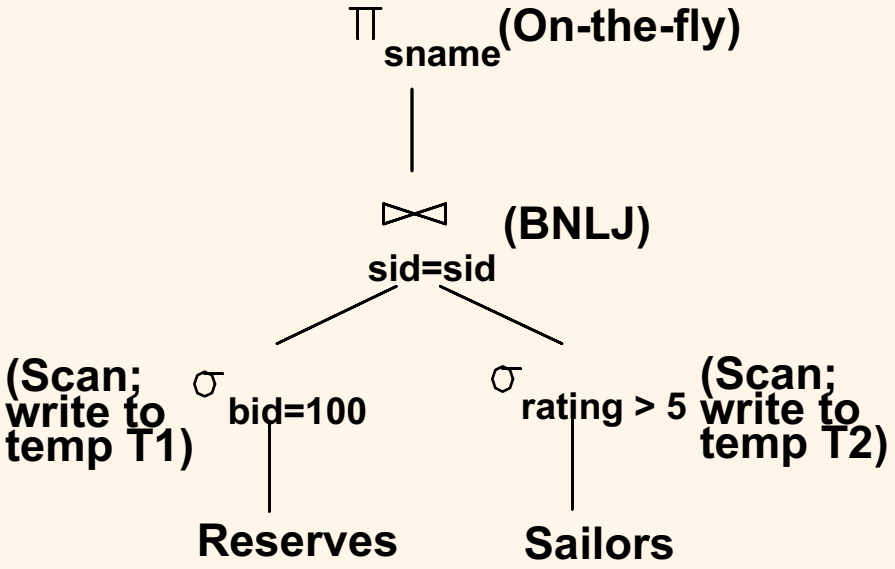
- Just means that we can apply the operation while the tuple is already in memory
- So no extra I/O cost

Alternative Plans (No Indexes)



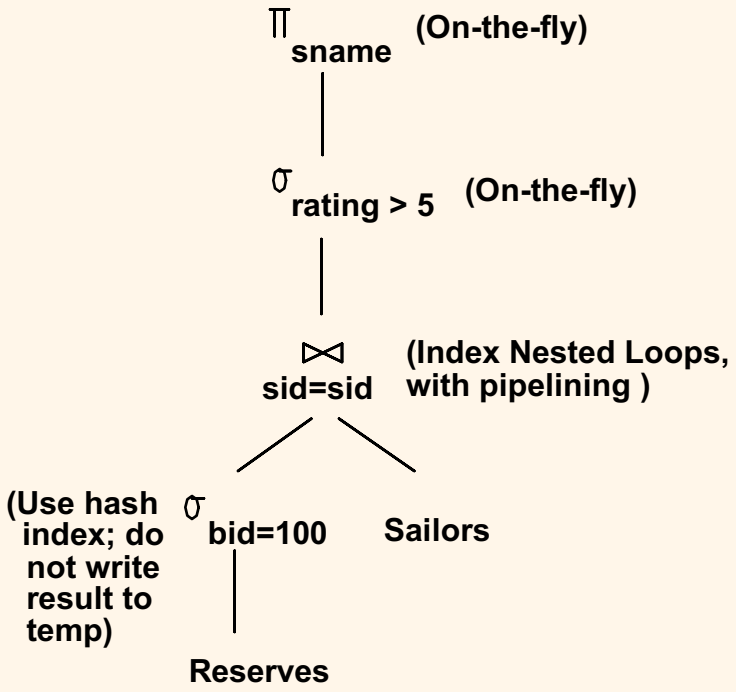
- Sort-merge join with 5 buffers:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os.

Alternative Plans (No Indexes)



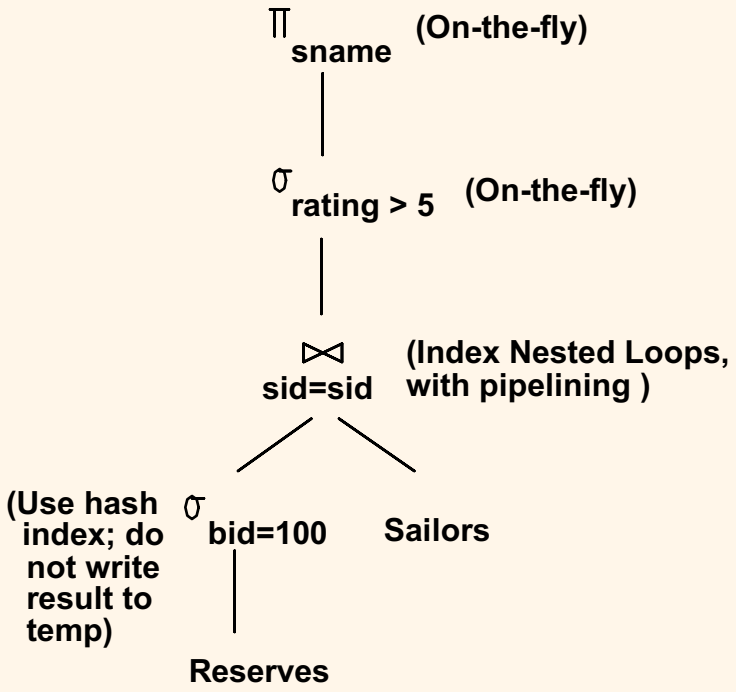
- Suppose we do BNLJ instead (3-pg blocks for T1) join cost = 10+4*250, total cost = 2770.
- If we push projections, T1 has only *sid*, T2 only *sid* and *sname*:
  - T1 fits in 3 pages, cost of BNL drops substantially, total < 2000.

Alternative Plans (With Indexes)



- Now suppose have hash index on bid of Reserves and a hash index on sid of Sailors
- INLJ with ***pipelining***
  - Outer relation in the join is never materialized
- Join column *sid* is a key for Sailors.
  - At most one matching tuple

Alternative Plans 2 With Indexes



- With clustered index on *bid* of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.

- Decision not to push *rating*>5 before the join is based on availability of *sid* index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

Calculating cost of a plan

- Cost components:
  - Reading input tables
  - Cost of each node/operator in the plan
    - Including writing intermediate tables if appropriate
  - Sorting result at the end if needed

Calculating cost of a plan

- Need to make assumptions about data to estimate size of intermediate tables and results
  - What fraction of the tuples will pass this selection condition?
  - How many tuples from relation R1 will join with each tuple from relation R2?

Estimating sizes

- How many tuples pass the selection in "SELECT * FROM R WHERE R.A < 10"?
- We want the reduction factor for the selection

  # result tuples = # tuples in R * reduction factor

- Several approaches depending on precision desired

Reduction factors

- Can pick an arbitrary reduction factor e.g. 0.1 (0.3 for inequality constraints like R.A < 42).
  - This may be enough! Remember we just want to compare plans to each other, not compute the true costs
- Can use a more precise reduction factor based on statistics/histograms

Data statistics

- DB catalogs typically contain at least:
  - # tuples and # pages for each relation.
  - # distinct key values and low/high key values for each index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) sometimes stored.

Reduction factors

- In our homework/exam questions we usually give you some relevant info about data distribution
  - If we do, use that info.
  - If we don't, assume some sensible reduction factor (and tell us what assumption you are making)

Reduction factors for joins

- SELECT * FROM R, S WHERE R.A = S.B
- How big is the join result?
  - Could vary from 0 to the product of |R| and |S|
- Various heuristics with various levels of precision
  - Your textbook discusses some of them (Section 15.2.1)
  - You'll explore some others in the Practicum P5
  - In an exam/homework question, watch for relevant info such as primary/foreign keys.

Enumeration of Physical Plans

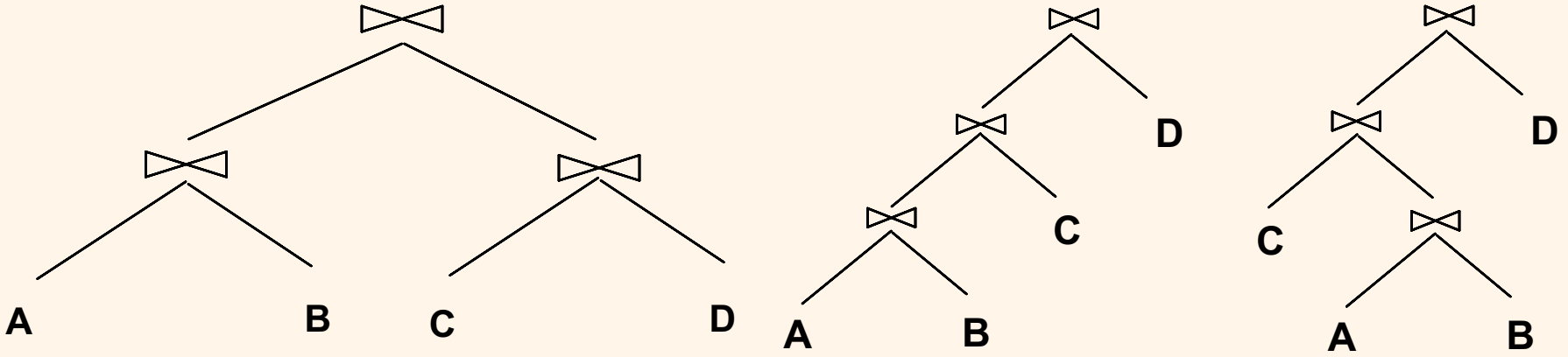
- There are two main cases:
  - Single-relation plans
  - Multiple-relation plans

Single relation queries

- No joins (by definition)
- Need to access the relation somehow
  - file scan or index
  - consider each possible access path and pick cheapest one
- Tuples are then pipelined to remaining selections, projections, aggregates.

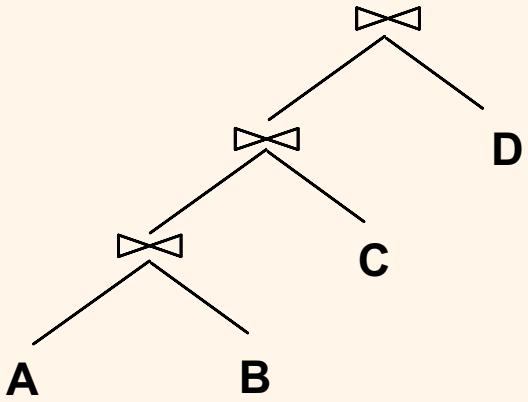
Queries Over Multiple Relations

- Core of the problem: how to evaluate the joins?
  - Can't consider all possible orderings
  - (Remember, if it takes you longer to optimize than to run the unoptimized query, that defeats the purpose….)



Queries Over Multiple Relations

- Decision: *only left-deep join trees* are considered.
  - Left-deep means the **right child of each node is a base table** (a real DB table, not a join)
    - (pushed selections/projections on base tables ok)



Queries Over Multiple Relations

- Considering only left-deep trees cuts down on search space
- Left-deep trees allow us to generate all fully pipelined plans (which are particularly desirable)
  - Outer input to join is never materialized
  - Not all left-deep plans are fully pipelined (e.g., plans that use SMJ).

Enumeration of Plans

- Left-deep plans differ only in the order of relations, the access method for each relation, and the join method for each join.

Finding the best plan

- Approach 1: exhaustive recursive search

Finding the best plan

- Consider the best plan for joining *k* relations
- There are *k* options for the last relation in the join
  - For each choice of the last relation, we want to join it with the optimal plan for the remaining *k-1* relations
- How to compute optimal plan for *k-1* relations?
  - Pick one as the last relation and recurse…

Finding the best plan

- Better approach: avoid recomputation through dynamic programming
- Save intermediate results that will be reused later
- Compute bottom-up from subsets of size 1, 2, 3, etc.

Enumeration of Plans

- Pass 1: Find best 1-relation plan for each relation
  - includes any selects/projects just on this relation.
- Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)*
- Pass k: Find best way to join result of a (k-1)-relation plan (as outer) to the kth relation. *(All k-relation plans.)*

Some practical remarks

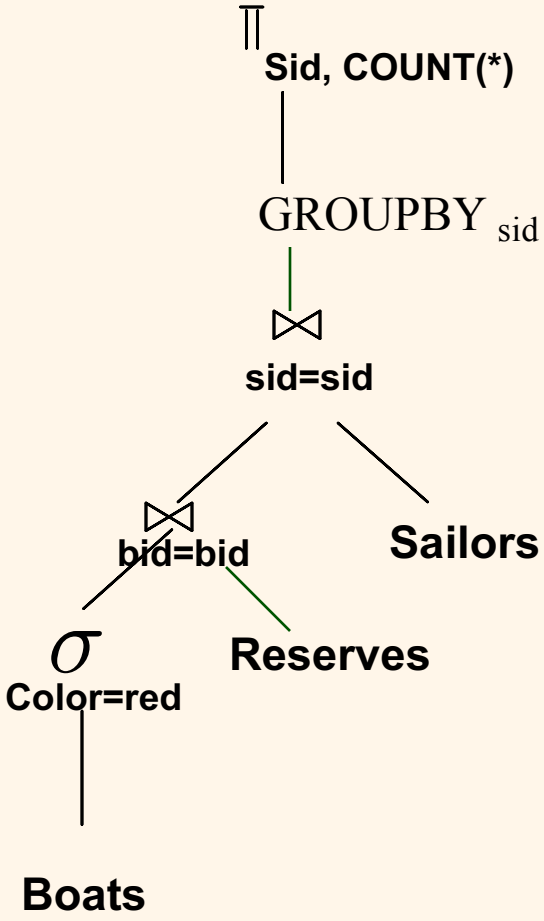
- Cost function for intermediate plans may be number of I/Os or something else
  - Simpler e.g. intermediate relation sizes
  - More complex e.g. include whether plan produces tuples in a sorted order (could be useful!!)
  - Your textbook presents an algorithm where we retain both the cheapest plan and any sorted-order plans

Enumeration of Plans (Contd.)

- ORDER BY, GROUP BY, aggregates etc. handled as a final step
  - Use a plan that gives result in sorted order
  - Or use an additional sorting operator

Example

Sailors:
  Hash, B+ on *sid*
Reserves:
  Clustered B+ tree on *bid*
  B+ on *sid*
Boats
  B+, Hash on *color*




```
SELECT S.sid, COUNT(*)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
GROUP BY S.sid;
```

Pass 1

- Best plan for accessing each relation regarded as the first relation in an execution plan
  - Sailors: File Scan
    - going through B+ index gives tuples in sorted order but index is unclustered – not a good plan (file scan + sort likely better!!)
  - Reserves: File Scan
    - clustered B+ index also works but why pay the overhead of root-to-leaf navigation?
  - Boats: Hash index on color

Pass 2

- For each of the plans in pass 1, generate plans joining another relation as the inner, using all join methods
  - File Scan *Sailors* (outer) with *Reserves* (inner)
  - File Scan *Reserves* (outer) with *Sailors* (inner)
  - *Boats* hash on color (outer) with *Reserves* (inner)
  - File Scan *Reserves* (outer) with *Boats* (inner)
  - … etc
- Retain cheapest plan for each pair of relations
  - Also sorted-order plans even if they are not cheapest

A pruning heuristic

- Do not combine a partial plan with a relation unless there is a nontrivial join condition between them
  - i.e., avoid Cartesian products if possible.
  - in our case, don't bother with the Sailors/Boats pair of relations
  - note may not always be possible (e.g. if query requires cross product)

Pass 3

- For each of the plans retained from Pass 2, taken as the outer, generate plans for join with the last table
  - E.g.:
  - Outer: Boats hash on color with Reserves (bid) (sort-merge)
  - Inner: Sailors (file scan)
  - Join algorithm: sort-merge

Add cost of aggregate/GROUP BY

- Cost to sort the result by sid, if not returned sorted

Optimization summary

- Parse query into RA tree
- Optimize one block at a time
- Generate a subset of the possible evaluation plans
  - Dynamic programming approach
  - Investigates only left-deep join plans
- Cost calculations based on statistics maintained in the catalog